ELIoT: Design of an Emulated IoT Platform

Alli Mäkinen, Jaime Jiménez, Roberto Morabito
Ericsson Research, Finland
{alli.a.makinen, jaime.jimenez, roberto.morabito}@ericsson.com

***Abstract*—The constant and rapid evolution of hardware and software components, along with the proliferation of newer standards and technologies, makes the creation of IoT environments devoted to test purposes extremely complicated. In this work, we present the design of an IoT emulation platform (ELIoT), which aim on making the deployment of IoT test environments easier and more flexible. ELIoT provides support for well-known IoT protocols — e.g., Constrained Application Protocol (CoAP) and Lightweight M2M (LWM2M) —, facilitating the study of interactions between different IoT entities. In order to make ELIoT easily portable and highly customizable, we exploit the lightweight and flexible characteristics of emerging virtualization technologies, such as Docker containers, for emulating both simple and complex IoT devices. A comprehensive performance evaluation aims to assess the feasibility of our implementation, by testing ELIoT scalability properties and other relevant performance metrics such as communication latency and throughput.**

I. Introduction

The constant and fast evolution of heterogeneous network and sensor technologies has led to the increase of the number of connected devices to the Internet, bringing new network deployment challenges [1] [2]. Machine-to-machine (M2M) solutions have been around for several decades enabling devices, such as sensors, to be connected to one another and to communicate on closed purpose-built networks. The Internet of Things (IoT) has brought the possibility to make heterogeneous devices communicate with each other by means of general purpose Internet protocols and standards in a loosely and secure fashion [3].

The majority of IoT devices have limited computing capabilities, memory resources, and battery life. IoT networks have to take into account such technical limitations, while maintain lightweight nature, and at the same time ensure constant reliability and security. The optimal design of IoT networks is therefore influenced by all these requirements that somehow affect the design decisions, such as protocol choice and security mechanisms to be featured [4]. Another important requirement is given by the need of flexible and adaptable communication models, which leverage the Representational State Transfer (REST) model. They have been adapted to the constrained space, together with additional Web key elements (such as uniform representation of documents and uniform identifiers), and used to make constrained devices part of the Web [5].

As an example, Constrained Application Protocol (CoAP) [6] was designed as universal transfer protocol with uniform identifiers for resources, methods for enabling resources interactions, resources discovery mechanisms, and security extensions. On top of CoAP, various resource representation formats arose. One of them,L WM2M Objects came along with the Lightweight Machine-to-machine (LWM2M) protocol [7] — which extended CoAP to a general purpose device management protocol. Based on the LWM2M objects, IPSO Alliance [8] further defined a set of general-purpose objects, which represent common type of sensors and actuators.

The interdependence between sensing, actuation, control and computational logic in IoT systems introduces challenges for application designers. As an example, deploying a testbed of thousands of devices in a real environment just for prototyping purposes is time-consuming and relatively expensive when compared to emulation [9]. Moreover, hardware-based prototyping systems tend eventually to become obsolete. In scenarios where the assumptions to be evaluated deal more with scalability, networking and software it may be more appealing, cost-efficient and faster to simply emulate those devices [10]. Therefore, emulation of IoT devices can play a key role on facilitating test and deployment of emerging IoT solutions and implementations.

By definition, emulated environments are not completely faithful to the real world, but they can provide a high-degree of confidence regarding the emulation of a particular scenario. Emulators use the exact same code, software, protocols and packets payload is virtually indistinguishable from those of the real devices. Moreover, a simple logic on the emulated node provides a more realistic simulation of the node's behavior.

There are several simulation environments developed for mimicking resource management decisions of different systems, such as cloud and grids [10]. One such solution applicable to IoT scenarios is SimIoT, a toolkit for experimenting on dynamic and real-time multi-user submissions. It features VM management, messaging based on real-time constraints and IoT inputs, but currently is missing for example support for heterogeneous IoT devices [10]. In addition, several wireless sensor network (WSN) simulators exist. For example Cooja [1] is a popular Contiki network simulator, which provides insights on network connectivity and protocol performance. However, like many other WSN simulators, it lacks support for modeling sensing coverage [11].

In this paper, we describe the implementation of an emulated IoT platform, that we refer here as ELIoT, using a full CoAP network stack and LWM2M/IPSO objects as representation formats. The emulated testbed allows simplifying and facilitating IoT-system testing, providing devices with logic and enables easy scalability test evaluations thanks to the possibility of emulating a large amount of devices. Compared to the aforementioned simulators, our ELIoT platform focuses on the device side, modeling the behavior and communication patterns of real devices, abstracting the networking within Docker containers.

The paper is organized as follows. Section II provides background information about CoAP, LWM2M and Docker, while Section III presents design requirements and current implementation of the ELIoT platform. Section IV shows the results of an extensive performance evaluation that aims to establish, by considering a specific IoT scenario, of ELIoT implementation impact on general purpose server machines. Section V concludes the paper and provides final remarks.

II. Background Information

The Constrained Application Protocol (CoAP), defined in [6], is a lightweight RESTful web transfer protocol designed for constrained devices and networks. CoAP redesigns HTTP adding specialized IoT properties such as low overhead, multicast support, and built-in resource discovery.

Built around CoAP, the Lightweight Machine-to-machine (LWM2M) protocol provides a series of interfaces for device management as well as a data model to express them [12]. Typically a LWM2M-based system will feature a *LWM2M Server* and *LWM2M Client*, even though in practice both act as client and server depending on the management function being used. A LWM2M Server is typically located in a private or public data center and can be hosted by the IoT service provider to manage devices [13]. The LWM2M Client is a software that resides on the device [14], hosting a CoAP Server too. An optional *LWM2M Bootstrap Server* can be used to manage the initial configuration parameters of LWM2M Clients during the device bootstrapping.

Management operations between Server and Client are grouped logically into four interfaces: *Bootstrapping*, *Device Discovery & Registration*, *Device Management & Service Enablement*, and *Information Reporting*. LWM2M also specifies a data model. The atomic components are called *Resources*. They represent basic parameters that map to specific values (e.g. Binding Mode, SMS Number, etc.) logically organized into *LWM2M Objects*. Those *Objects* represent normative management units [7]. Alternatively from the management domain, other Objects have been defined to build applications. For example, the IPSO Alliance has defined a set of general purpose Objects [15], which represent common sensors and actuators. These Objects can be used agnostic of the application protocol [16].

We consider the ”Client Initiated Bootstrap” as defined in [7] as our typical bootstrapping scenario. In such scenario, each LWM2M Client requests the bootstrap information from a LWM2M Bootstrap Server and registers several distinct objects to a LWM2M Server. Upon registration, the Server assigns an unique endpoint ID to each Client. Once the device is registered, the LWM2M Server is able to perform operations, such as read, write and execute, on the device’s Objects and Resources using standard CoAP method calls (GET, PUT, POST, DELETE).

Docker container-based virtualization has been the software technology used for enabling the emulation of IoT devices [17]. The reasons behind the choice in this context are multiple. Mainly is the fact that Docker gives the possibility to ensure an easy cross-platform deployment. Indeed, container technologies allow to deploy reusable software without dependency on heterogeneous hardware devices. This means that ELIoT can easily run, as long as Docker is supported, in a machine with rather different hardware architectures.

A Docker container is a runnable instance of a Docker image. When a container is created/executed, the configuration setup of the *dockerized* application is specified together with any other dependency (e.g., libraries). Containers package the application with all the necessary runtime dependencies, system tools and libraries, and isolate it from other applications and the underlying infrastructure. While running a container from an image, Docker uses an overlay file-system (UnionFS) to add a read-write layer on top of the image. This speeds up the building process and saves disk space compared to alternative virtualization solutions [18].

The Docker network-setup may represent a key aspect when ELIoT users want to create their own emulated scenario. By default, Docker creates a *docker0* interface as an Ethernet bridge on the host system typically with a subnet 172.17.0.0/16. Each running container features specific MAC and IP addresses from docker0 addresses pool. However, the *bridge* mode has an upper-bound on the number of ports — 2ˆ10 (1024) — that the host can map into running containers [19]. To overcome this limitation, the network-setup can be set to e.g. *host*. In the *host* mode each attached container uses the hosts network stack. This implies that there is no isolation between containers and host machine, which may result

[1] http://www.contiki-os.org/start.html

TABLE I
LWM2M DESIGN REQUIREMENTS

Functionality	**Description**	**Pre-ELIoT**	**After-ELIoT**
Bootstrapping	LWM2M client can perform device bootstrapping	**No**	**Yes**
Registration/ Deregistration	LWM2M client can register/deregister to/from LWM2M server	**Yes**	**Yes**
Read/Write/Execute	LWM2M client can read/write/execute resources	**Yes**	**Yes**
Create/Delete	LWM2M client can create/delete object instances	**No**	**Yes**
Observe/Notify	LWM2M client supports observe/notify concept	**Yes**	**Yes**
JSON/TLV/CBOR	LWM2M client supports JSON/TLV/CBOR formats	**Yes/No/No**	**Yes/Yes/No**
D2D communication	LWM2M client can communicate directly with other clients	**No**	**Yes**
DTLS security	LWM2M uses DTLS based security	**No**	**No**

in security issues. However, such setup improves networking performance, as it removes the overhead due to the virtualization of the network layer.

III. PROPOSED DESIGN AND IMPLEMENTATION

ELIoT makes use of standard Internet protocols for emulating individual devices and various interaction patterns between them. We not only use ELIoT to perform stress tests on management systems, but we aim on investigating device-to-device (D2D) interactions using the same standards.

As already discussed before, deploying a testbed with physical sensors and actuators can represent a costly and not efficient effort. For this reason, by taking into account our requirements and the needs of emulating a large amount of devices, we decided to utilize container-based virtualization — and in particular Docker containers — for equipping our system-emulator of a software tool that can allow to get rid of cumbersome testbed setup operations, and enable simpler scalability evaluations.

In ELIoT, an emulated device, a dockerized application, consists of one or more virtual sensors or actuators that can interact with other devices (emulated or not) and services on the Internet. Emulated devices act exactly as the equivalent physical one (e.g. humidity sensor or light switch) providing the associated measured data, or triggering an action according to a specific value provided by the sensor itself. Based on the specification, Table I shows the main functional requirements for a complete LWM2M environment, their implementation status within already existing projects, and the contribution that ELIoT has provided. As an Open Source project, ELIoT aims to fulfill all the design requirements [20].

Aside from the functional requirements, scalability represents also a key aspect of our implementation. As the number of connected devices increases, bandwidth and memory usage may become a problem, if not properly considered in the implementation. Therefore, based on the fact that each emulated device corresponds to a Docker container, reproducibility of containers, communication latency and throughput are crucial performance aspects that have to be considered on assessing the goodness of our implementation.

The device logic is implemented on the device application, which in turn emulates the behavior of the real devices. The ELIoT platform is built on top of a java-based LWM2M open source implementation called *Leshan*[2] for the server side, and *coap-node*[3], which is a nodejs implementation of LWM2M for the client side.

As we focus on emulated devices, we wanted to utilize a *LWM2M Server* module that is already widely used and supported by the IoT community. This has led to the choice of Leshan, which provides support for CoAP, DTLS, LWM2M and IPSO objects, thus representing a strong base for fulfilling the design requirements mentioned in the previous section. Leshan project provides also demo-servers (management & bootstrapping) with simple Web user interfaces supporting basic server-side functionalities [21].

The coap-node module supports CoAP and LWM2M protocols along with the IPSO model, providing several APIs for building devices and handling the client/server interactions through LWM2M interfaces [22]. Existing functionalities of the coap-node implementation have been widely tested in the past. This gives us a reliable starting point for building our own LWM2M client applications for the emulated devices.

The integration between Leshan and coap-node has required minor modifications to the existing code, as coap-node is originally designed to be used with a different nodejs LWM2M Server implementation. In addition, we have contributed to the module by implementing new APIs to support *Bootstrap* interface along with other missing LWM2M operations, such as *Create* and *Delete*, which have the crucial role of adding and removing object instances. Also support for TLV data format is added.

The need for D2D communication arose from the IoT use case we have considered and that will be introduced later in the paper. However, LWM2M does not provide methods for the devices to discover services and resources around them. For this reason, we have implemented D2D communication beyond LWM2M scope. In the long run, implementing a separate CoAP Resource Directory (RD) based on [23], with fully support for ”well-known locations” can represent an important implementation enhancement as would make ELIoT aligned with ongoing standardization efforts. However, the use of CoAP multicast can well match with our needs. Indeed, CoAP multicast is convenient in group communications, where a message needs to be sent to multiple participants, like in our use case. However, it may be too heavy solution, when the communication takes place only between a few devices.

[2] https://github.com/eclipse/leshan

[3] https://github.com/PeterEB/coap-node

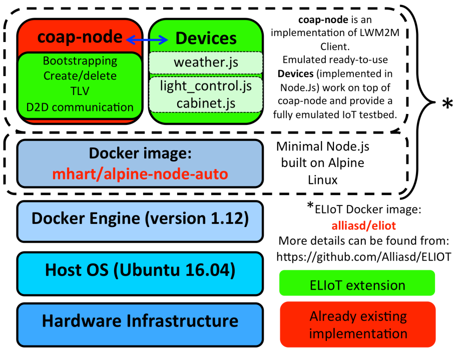
coap-node
Bootstrapping
Create/delete
TLV
D2D communication
Devices
weather.js
light_control.js
cabinet.js
coap-node is an implementation of LWM2M Client.
Emulated ready-to-use **Devices** (implemented in Node.Js) work on top of coap-node and provide a fully emulated IoT testbed.
*
Docker image:
mhart/alpine-node-auto
Minimal Node.js built on Alpine Linux
Docker Engine (version 1.12)
Host OS (Ubuntu 16.04)
Hardware Infrastructure
*ELIoT Docker image:
alliasd/eliot
More details can be found from:
https://github.com/Alliasd/ELIOT
ELIoT extension
Already existing implementation

Fig. 1. Structure of the ELIoT stack.

The ELIoT platform provides ready-to-use devices, such as weather observer, presence detector, light controller and radiator. The implementation is open source under MIT license and available in the following GitHub repository: https://github.com/Alliasd/ELIoT. In such repository, it is also possible to find all the configuration files needed to setup the emulated devices — by means of Docker containers — together with detailed instructions on how to run different emulated devices.

In ELIoT, we have created our own Docker image to store all the device applications, together with the entire configuration files needed to run the emulated devices, as shown in Fig. 1. Each container represents an emulated device. In this way, it is easier to characterize system resources and performance metrics of each emulated sensor, which will benefit also of an own IP address. The device emulation is based on our customized Docker image *alliasd/eliot*[4]. Starting from this Docker image, the instantiation of Docker containers generates, in practice, fully functional emulated devices. For the LWM2M server emulation, we utilize a public Leshan Docker image *corfr/leshan*[5].

IV. Evaluation and Testing

In this section, we evaluate functionalities and scalability of ELIoT. The evaluation of the functionalities is based on the execution of our use case. More specifically, it analyzes the fulfillment of ELIoT design's requirements, as well as the remaining design issues that are left for future implementation. The scalability is evaluated in different environments in terms of reproducibility and activation time of containers, communication latency and throughput.

A. Use case

In order to test ELIoT performance, we reproduce an IoT use case referred as *public lighting scenario*. The system emulates lighting points and a control cabinet.

TABLE II
Requirements for Public Lighting

Bi-directional communication	**Required**
Data collection mode	Periodic + alerting
Period of collection	one to a few / day
Tolerated latency for collection	1 minute

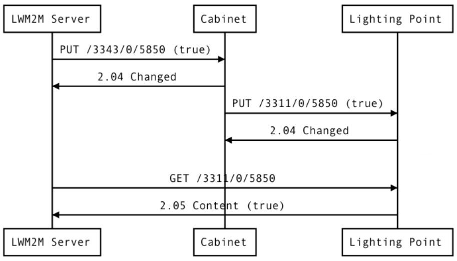

Fig. 2. Communication logic of public lighting.

This use case was chosen as it shows an example of using simple sensors collecting information from the environment and actuators that trigger switching actions. The interaction among multiple devices occurs through multicast communication.

The lighting system measures illuminance and control light switches. Typically, sensing operations take place at street cabinet, which supplies several lighting points. Each lighting point has control unit, which is controlled in terms of time and light level in the evenings. ETSI has defined a set of requirements for the public lighting scenario [24], that are presented in Table II.

Network communication mechanisms, functionalities, and devices process operations are referred below as *communication logic*. The communication logic of public lighting is fairly complex. After the lighting points are registered to the LWM2M server, they can be passively monitored or individually controlled — by changing their status or dimming settings — on the LWM2M Server itself. However, the main control is done through a dedicated street cabinet. When the cabinet's timer is turned-on from the management server, it waits until the next execution event is reached. The event triggers the switching and dimming of lighting points according to the current illuminance level or time. Fig. 2 shows the communication sequence for the aforementioned scenario.

B. LWM2M functionalities evaluation

The basic LWM2M functionalities are implemented in accordance with the LWM2M specification, but some more advanced capabilities for D2D communication are still under development. The table I shows which LWM2M functional requirements are fully/partially fulfilled. ELIoT features that are still not fully available or implemented are also listed.

[4] https://hub.docker.com/r/alliasd/eliot/

[5] https://hub.docker.com/r/corfr/leshan/

The *bootstrap* represents the first operational step to be executed for the setup of emulated devices, and it is completed via the Bootstrap Server. In order to accomplish this operation, the device application has to be started by setting the *bootstrap flag*. The ELIoT platform provides a bootstrap data file shown below, which defines the server and security objects required to find the management server and configure the basic device settings. The values can be changed according to the use case. From a security point of view, currently there is not a complete implementation of DTLS for nodejs available, and therefore a non-secure communication is used.

```
{
    "servers": {
        "1": {
            // Device Identifier
            "shortId": "1",
            // Min and Max Observation period
            "defaultMinPeriod": 10,
            "defaultMaxPeriod": 60
        }
    },
    "security": {
        "1": {
            // CoAP Server URI
            "uri": "coap://ms:5683/",
            "bootstrapServer": false,
            "securityMode": "NO_SEC",
            "serverId": "1"
        }
    }
}
```

Once the bootstrapping is finished, the emulated devices are aware of the correct LWM2M Server address, so they can automatically *register* to it. During the registration phase, the devices send to the LWM2M Server all the LWM2M and IPSO objects and other mandatory information — such as *endpoint name* and *lifetime value*. All of our emulated devices have the same LWM2M objects: server, security, device, location, and connection monitoring. In addition, timer and illuminance IPSO objects can be associated with the control cabinet, and light control IPSO object with lighting points.

The current implementation allows *reading, writing, executing* and *observing* any defined resources. In general, the single-value data (resources) is represented in text, JSON or TLV format, whereas multi-value data (object instances) is represented in JSON or TLV format.

In the public lighting use case, the *D2D communication* is required. The control cabinet sends a multicast message with appropriate payload (ON/OFF) to the lighting points twice a day. Such messages are sent to all participants listening the CoAP multicast address, which may cause tangible of overhead in some cases, but it is an appropriate solution here, as there is one central control unit and multiple endpoints, and reliability is not required.

In addition, LWM2M *create* and *delete* operations are partly implemented. One can create new object instances and delete them, but delete operation has some special cases, which are not implemented yet.

The emulated devices, can be also *deregistered* by sending a *delete* message to the LWM2M server, once the applications execution is completed.

C. Scalability Evaluation

In IoT platforms, mass registration of IoT devices must be supported and it represents a key requirement to be fulfilled. This requires that the Server is able to simultaneously manage thousands of clients, which have to be able to reach the server in a relatively small time frame. A greater time spent on setup and register a device, implies that the platform's resources will not be available for the execution of other tasks, producing also an ineffective increase in the system load of the platform.

In the case of ELIoT, three relevant aspects have to be considered in order to verify that the emulator is efficient on meeting the above mentioned requirements: (i) define the upper-bound for the number of Docker containers that can be simultaneously executed, (ii) quantify the activation time of one or multiple containers, (iii) verify that the latency among request and response of a device is compliant to real IoT use cases. With regard to the first requirement, there is clearly a strong dependency on the host — and the underlying hardware — used for the test. However, our goal is to be able to run simultaneously as many containers as possible, by considering scenarios with physical and virtual machines.

The reproducibility of the devices is the essential benefit of the ELIoT platform. To get an understanding of how well ELIoT scales up, the first phase of our scalability testing was to find a maximum amount of containers on a single physical host using Dell Latitude E6540 machine with Intel Core i7-4800MQ CPU (2.70GHz, 8 cores) and Ubuntu 14.04 Operating System. It has 16GB of memory and 512GB Disk.

In order to be able to launch simultaneously multiple devices, we used *Docker-Compose*[6], which is a tool for defining and running multi-container Docker applications. In Docker compose, the *scale* option allows to set the number of containers to run for a service. The unix tool dstat [7] has been used for measuring the *system load* of the hosts in which the scalability tests are performed. The system load average is a metric that indicates the overall amount of computational work performed by a system, and it is expressed over a period of time. As our test machine features an 8-core CPU, the system load upper bound is eight. Overcoming such upper bound means that some of the running processes need to wait for their execution, by consequently producing a general system performance degradation. The more the upper bound is exceeded, the more the system operations are slowed down.

[6] https://docs.docker.com/compose/

[7] http://dag.wiee.rs/home-made/dstat/

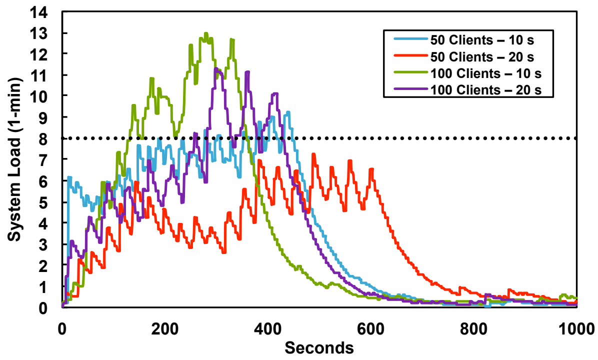

Fig. 3. 1-minute load average of the host machine over 1000s.

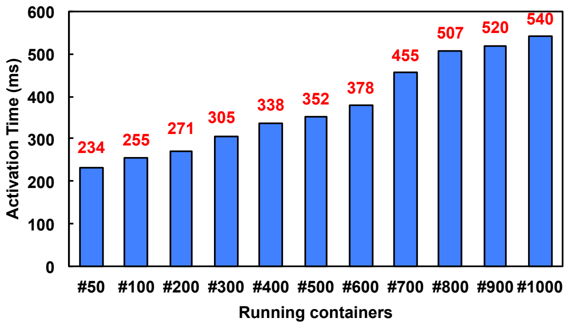

Fig. 4. Containers activation time.

In the first scalability test, we used the bridge mode as network setup, and scheduled the launch of fifty *weather observer* devices every ten seconds. The ten seconds delay was necessary to avoid HTTP timeout errors from the Leshan Web Interface. As expected, as a result of the use of this network setup, it has been possible to run a maximum of 1023 containers (1 server, 1022 devices). Load average was below ten, exceeding the upper-bound within reasonable limits.

However, as the system load is a common limiting factor, we wanted to study further, how it affects the performance of the platform. We compared the results of (i) scheduling the launch of 50 devices every ten and (ii) 20 seconds to (iii) scheduling the launch of 100 devices every 10 and (iv) 20 seconds. Fig. 3 shows the system load average over 1000 seconds period, as 1000 devices were launched according the aforementioned setups. It is clear that scheduling less devices with longer idle time results in more *stable* performance — only setup (ii) achieves a load average below the limit eight, but the registration process of the devices is the longest, 600 seconds. On the contrary, the setup (iii) provides the highest load average and thus, the most *unstable* performance, which can be seen as larger variance of load average above the limit. However, the overall registration time of the devices is the fastest, 320 seconds. An interesting remark is that the outcomes of setups (i) and (iv) are very similar. The registration times are 435 seconds and 415 seconds respectively, but setup (iv) introduces higher system load. It seems that higher number of devices requires comparatively more idle time.

The network limitation can be overcome by using the *Docker host-network setup*. In our case, to exceed the 1023 containers, we would also need more memory and processing power. For that reason, we repeated the scalability test on Azure cloud environment with 50 virtual machines (VM). The VMs are part of Azure DSv2-series featuring a 1-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor, 3.5 GiB of memory and 7 GiB local SSD. As the VMs do not have as much resources as the physical machines, using the Docker bridge network with the container limitation was sufficient. One VM was dedicated only for the Leshan server, while others were running clients, simple *presence detector* devices.

With this setup, we were able to run 7123 devices in total. This was achieved by having 150 containers per VM. It took 109 seconds to register all the clients to the LWM2M Server. In terms of memory the VMs should have been able to handle around 250 containers in theory. The Azure interface showed some momentary CPU spikes as we launched multiple devices simultaneously, but even when the devices were launched in a slower tempo, we hit the same limit.

In all of the previous test scenarios, it was clearly noticeable that the process of registering devices slowed down in time with the increasing amount of containers. To examine the issue more thoroughly, we calculated the *activation time* of a Docker container. *Activation time* describes, how long it takes for a container to reach a running state after starting it. The test was conducted by starting fifty *light controller* devices, and increasing the device allocation by 100 until 1000 devices was reached.

Results are shown in Fig. 4. As we can see, the activation time increases gradually from 234 ms to 540 ms, when the amount of containers running in background increases. Thus, the process of connecting the devices to the Server clearly slows down linearly.

Like the activation time increases with the growing number of containers, so does the latency between the sent requests and received responses. We additionally did some measurements regarding throughput (requests per second) and latency (time between requests and responses) to see, how much we can stress our devices. For these tests, we used a special test client, a temperature sensor, which sends registration updates to the LWM2M Server over 5-10 minutes time period depending on the amount of clients.

The measurements were done with two setups: with and without docker-compose. With Docker-compose, the containers were created and started simultaneously

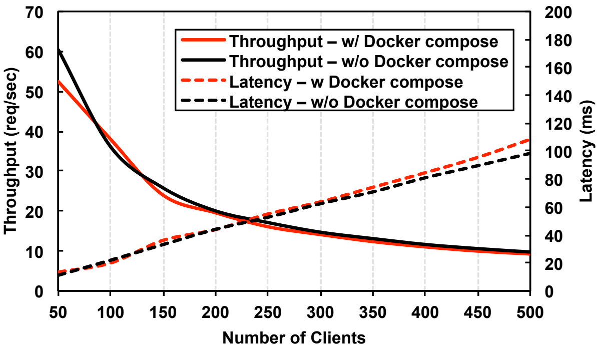

Fig. 5. Throughput and Latency evaluation.

during the scaling process, whereas without docker-compose, the containers were started once all the containers were created first. In both scenarios, the requests were scheduled to be executed after a minimum threshold of 5 milliseconds due to the asynchronous nature of nodejs. A smaller delay with docker-compose caused the system to become too busy. We first scheduled the launch of 50 clients and increased the number by 50 until we reached 500 clients.

The results are shown in Fig. 5. It shows the average number of requests per second produced by a client and also the average latency per request. The latency increases linearly, while the throughput decreases fast from 50 to 150 clients and then slowly. As expected, latency performance is slightly better without docker-compose; the creation time of a container is excluded from the results.

V. Conclusion

In this paper we presented the ELIoT platform, an Emulated IoT testbed that can be used for building and managing emulated CoAP-enabled devices and testing different IoT communication patterns and protocols. ELIoT consists of LWM2M Server, Bootstrap Server and ready-to-use devices, which support basic LWM2M operations and IPSO objects, and feature simple device logic.

We presented a use case regarding public lighting, to demonstrate device management functionality, and D2D interactions with ELIoT platform. The use case involved simple sensors and actuators supporting bootstrapping, registration/deregistration, reading/writing resources, and several data formats, such as JSON and TLV.

ELIoT also aims for scalability. Thus, we conducted several experiments to test the reproducibility of thousands of emulated devices in different hardware environments. We have also evaluated how device emulation affects containers activation time, communication latency, and throughput.

Acknowledgment

This work is partially funded by the FP7 Marie Curie Initial Training Network (ITN) METRICS project (grant agreement No. 607728).

References

[1] A. Al-Fuqaha and others. " Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications", IEEE Communications Surveys and Tutorials. Vol7, 2015.

[2] L. Atzori and others. "The Internet of Things: A survey", Computer Networks, 54:15, October 2010.

[3] J. Hller and others. "From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence", UK: Elsevier, 2014.

[4] H. Tschofenig and others. "Architectural Considerations in Smart Object Networking", IETF RFC 7452, March 2015.

[5] A. Keranen, M. Kovatsch. "RESTful Design for Internet of Things Systems", keranen-t2trg-rest-iot, October 2015.

[6] Z. Shelby and others. "Constrained Application Protocol (CoAP)", IETF RFC 7252, June 2014.

[7] Open Mobile Alliance, "Lightweight Machine-to-Machine Technical Specification V1.0 Approved", February 2017.

[8] IPSO Alliance, https://www.ipso-alliance.org/, Accessed May 2017.

[9] G. Kecskemeti and others. "Modelling and Simulation Challenges in Internet of Things",

[10] S. Sotiriadis and others. "Towards Simulating the Internet of Things", 28th International Conference on Advanced Information Networking and Applications Workshops, May 2014.

[11] L. Sitanayah, and others. "Demo Abstract: A Cooja-Based Tool for Maintaining Sensor Network Coverage Requirements in a Building", 11th ACM Conference on Embedded Networked Sensor Systems (SenSys13), November 2013.

[12] J. Prado. "OMA Lighweight M2M Resource Model", IAB IoT Semantic Interoperability Workshop 2016, March 2016.

[13] S. Rao and others. "Implementing LWM2M in Constrained IoT Devices", Proceedings of Wireless Sensors (ICWiSe) IEEE Conference, August 2015.

[14] G. Klas and others. Lightweight M2M: Enabling Device Management and Applications for the Internet of Things", White Paper, February 2014.

[15] IPSO Objects, http://ipso-alliance.github.io/pub/, Accessed September 2016.

[16] J. Jimenez and others. "IPSO Smart Objects", IAB IoT Semantic Interoperability Workshop 2016, March 2016.

[17] Docker Inc, https://docs.docker.com/get-started/, Accessed January 2017.

[18] R. Morabito and others. "Hypervisors vs. lightweight virtualization: a performance comparison." Cloud Engineering (IC2E), 2015 IEEE International Conference on. IEEE, 2015.

[19] S. Seelam. https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems/, Accessed January 2017.

[20] ELIOT Project, https://github.com/Alliasd/ELIOT, Accessed September 2016.

[21] Eclipse Leshan, https://github.com/eclipse/leshan, Accessed September 2016.

[22] Client node of lightweight M2M (LWM2M), https://github.com/PeterEB/coap-node, Accessed September 2016.

[23] Z. Shelby and others. "CoRE Resource Directory", draft-ietf-core-resource-directory-10, March 2017.

[24] European Telecommunications Standards Institute (ETSI). "Electromagnetic compatibility and Radio spectrum Matters (ERM); System Reference document (SRdoc): Spectrum Requirements for Short Range Device, Metropolitan Mesh Machine Networks (M3N) and Smart Metering (SM) applications", ETSI TR 103 055 V1.1.1, September 2009.